

Middleware Lab 04 :

Mobility model implementation for more realistic simulations

Guillaume-Jean Herbiet
<guillaume.herbiet@uni.lu>

November 28, 2008

Abstract

During the previous labs, you implemented a complete protocol model for ad hoc instant messaging in OMNeT++ , an event-based discrete network simulator. Then, you instrumented your simulation code in order to collect statistics about this protocol so as to assess its performance. You also learned how to easily run a complete simulation campaign, and to analyze and consolidate the collected data.

In this lab, you will bring your simulation environment closer to reality by implementing a more elaborated mobility model, that tends to mimic human behavior in the real world.

1 Mobility in network simulation

For results from a network simulation to be trustworthy, it is necessary for the simulation environment to be as close as possible to the reality. In particular, it is important for the model of users mobility to be carefully designed and to be meaningful (i.e. representing a plausible mobility pattern for the users in the considered use case).

The types of mobility models can be classified as follows:

- **traces models** that replay previously recorded traces in the simulation environment;
- **synthetic models** that try to reproduce realistic behavior of mobility without the use of traces. Synthetic models use most of the time a random or probability-based behavior. They are of two kinds:
 - **entity models** where each node movement is independent;
 - **group models** where nodes movements are correlated. Group models are often composed of a group component and a random additional individual motion.

The model we will implement is a synthetic entity model.

Would you be further interested by this topic, I recommend you the following readings:

- **A survey of mobility models for ad hoc networks**, available at http://www.cse.iitb.ac.in/~varsha/allpapers/wireless/adHocModels/mobility_models_survey.pdf
- **Random waypoint considered harmful**, at http://www.comsoc.org/confs/ieee-infocom/2003/papers/32_03.PDF

2 The Hotspot mobility model

The mobility model we will implement is called **Hotspot mobility model** as it mimics the behavior of users moving from a point of interest to another and stopping for a certain time at each point of interest before moving to the next one.

This model is based on the **Human mobility model** introduced by Luc Hogue in his thesis called **Modelling, Simulation and Broadcast-based Applications**.

2.1 Human mobility

Humans do not move randomly. When moving, they have a determined target spot and move towards it. The target may be a few meters away (next shop, next crossroads, other sidewalk, etc) as well as far away (next district, next city, etc). Upon time, their target changes. Most of the time, people have a dynamically changing list of targets. This list of target spots is dynamically changing because of various parameters that will appear upon time (locations of the target places, closure times, high frequentation times, etc).

2.2 Mass mobility models

This pattern of mobility is intended to model node movement during which the nodes have momentum, and thus do not start, stop, or turn abruptly. Therefore, angle and speed variation are bounded. This tends to make the mobile nodes behave as it would in the real world.

When a host hits a wall, it reflects off the wall at the same angle.

2.3 Hotspot mobility model description

The hotspot mobility model defines that the simulation area is a bounded area. This area contains a set S of hotspots which act as intermediate destination for the nodes. A hotspot is located by its x, y coordinates. It is a round area defined by the value of its radius. Spots should not overlap. Nodes can be located either in or out of spots.

The mobility model used by the node which move out of a spot is a variant of the random walk mobility model which respects the mass mobility constraints.

On the first hand, when a node n is located out of any spot, it must choose a destination spot $d(n)$. To do this, it maintains a set $VS(n)$ of the spots it visited in the past. A node cannot visit twice a same spot. $d(n)$ is chosen so as it is the closest spot which has not yet been visited (which does not belong to $VS(n)$).

The node then moves towards its destination spot, with a certain degree of random variation in its direction. The angle range a node can choose from is bounded

by the two tangent lines to the spot passing by its current position and the direction is adjusted at every move step.

Those requirements ensure that the node will eventually reach its destination spot at one moment.

On the other hand, when the node n is within a spot $d(n)$, it moves according to a fixed direction mobility model. The current spot $d(n)$ is added to $VS(n)$ when entering, the node pauses for a random bounded period and a new destination spot is defined. After the pause time, the node starts moving in straight line to the center of the next spot.

If all spots have been visited ($VS(n) = S$) then $VS(n)$ is emptied and the search for a new destination is triggered again.

After each pause time, the node starts moving from speed 0, and increases its speed with a bounded variation at each movement step.

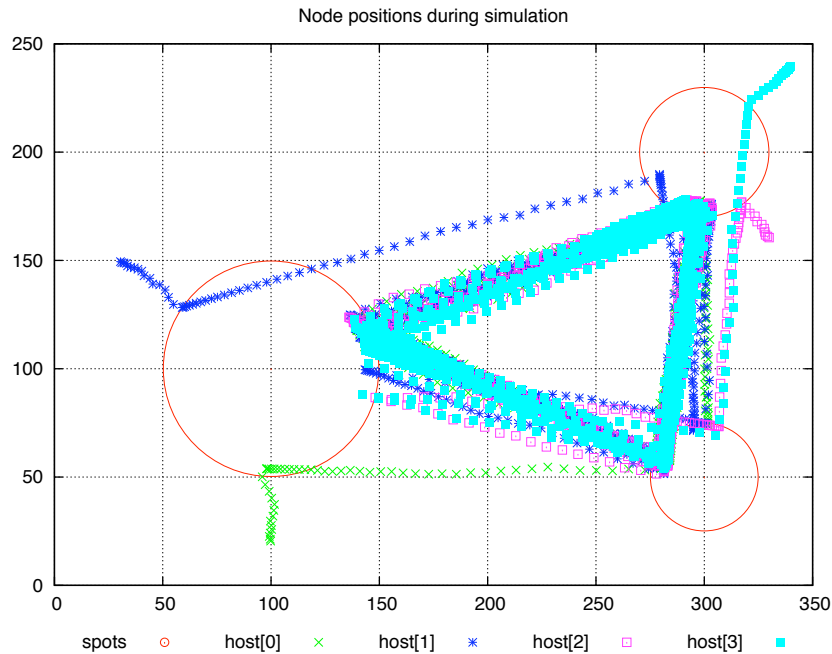


Figure 1: Example of node positions with hotspot mobility model

3 Mobility model implementation

3.1 Objectives

In this lab you will implement the hotspot mobility model in the OMNeT++ simulator by subclassing the Mobility Framework BasicMobility module.

Once this model will be implemented, you will use it in a simulation campaign and analyze the protocol behavior with this new mobility pattern.

3.2 The hotspots file

The list of hotspots and their characteristics (position of the center, radius) are described in a .spots file.

The principle used is the following: the channelcontrol module, upon initialization will open the .spots file whose filename has been given as module parameter and store the read spots in a list of HotSpot instances.

The HotSpot class definition is available in the /home/user/Development/templates/lab04 directory.

This modified channelmodule also performs a check of all the read spots before storing them: a spot should lie completely inside the simulation boundaries and not overlap with a previously defined spot. All spots not complying with this rule should be ignored.

You will be provided a template of a modified channelcontrol module called HotSpotsChannelControl that performs the hotspots reading and storing. The only things you will have to do are the following:

- integrate this new module type in the network model (it will replace the standard channelcontrol) and modify the .ini file consequently;
- implement the hotspot check in the HotSpotsChannelControl module;
- add a pointer to the HotSpotsChannelControl and read the list of correct spots at HotSpotsMobility module initialization.

The template and examples of .spots file are located in the /home/user/Development/templates/lab04 directory.

3.3 Mobility model algorithm

The following pseudo-code summarizes the intended behavior of a node following the hotspot mobility model:

Listing 1: Hotspot mobility model algorithm

```
Step 0: Chose the first spot
Step 1: Estimate angleMin and angleMax
3 Step 2: For each movement step
    Step 2.1: Pick currentAngle in [angleMin;angleMax]
    Step 2.2: Pick currentSpeed in [speedMin;speedMax]
6 Step 2.3: Compute position at end of movement step and move to position
    Step 2.4: If new position is in spot
        Then go to step 3
9        Else go to step 2
Step 3: Pause for a random bounded period of time
Step 4: Pick the next spot to visit
12 Step 5: Determine currentAngle as the direction to the next spot center
Step 6: For each movement step
    Step 6.1: Pick currentSpeed in [speedMin;speedMax]
15 Step 6.2: Compute position at end of movement step
    Step 6.3: Move to position
    Step 6.4: If new position is outside spot
18        Then go to step 1
        Else go to step 6
```

3.4 Mobility management with the Mobility Framework

The Mobility Framework allows each mobile host to follow a completely different mobility model. However, in most cases, they will all use the same. Therefore, in the Host model, there is a module called `mobility` in which you can implement your movement pattern. All mobility modules extend the `BasicMobility` module.

3.4.1 Position storage

Nodes are located on the playground using `Coord` objects, which is a set of two `double` attributes, respectively storing the position on the x-axis and on the y-axis. `Coord` objects have a `distance()` method that takes another `Coord` as argument and returns the euclidian distance between the object instance given as parameter and the object instance from which the method was called.

The host current position is stored in the `move` variable which is of type `HostMove`. This class has the following attributes that need to be updated at each movement step:

- `startPos` (of type `Coord`) that stores the node current position;
- `speed` (of type `double`) that stores the node current speed;
- `startTime` (of type `simtime_t`) that stores the time at which the current position was reached.

Besides, when the node is changing final destination, the method `setDirection()` of the `HostMove` class should be used.

3.4.2 Node movement

As long as the `speed` attribute of the `move` object is different from 0, a self-message is scheduled every `updateInterval` to trigger the node movement. The message is first scheduled in the stage 0 of the `initialize()` method, so ensure that each node has some non-null speed at that time or it won't start moving.

Each time this self-message occurs, two functions are called:

- the `makeMove()` function, that will compute the new position of the host, and update the `move` variable. This is where you will have to do all your computation and most of your work;
- the `updatePosition()` function, that will update the new host position at the `blackboard` module. This function is required for the `channelcontrol` to know the most recent position of the host in order to compute correct propagation information.

Practically, you will only have to redefine the `makeMove()` function to fulfill your needs.

Note that the host movement depends on whether or not the host is inside a spot. Such a behavior can be described as the host being in different states. So, here again, you can apply the finite-state-machine paradigm to design and implement the mobility pattern.

3.4.3 Reflexion at simulation boundaries

There are different way to model the fact that a node reaches the simulation boundaries:

- **random repositionning**: the node is placed at a random new position inside the playground. This doesn't represent any real behavior;
- **warping**: a node reaching a boundary will reenter the playground at the opposite position. This simulates a borderless simulation playground but may generate unrealistic effects;
- **reflection**: the node is reflected (or bounces) on the simulation border, like if a solid wall prevented the nodes for going away.

The latter solution is the most common and, thus, we will use this one in our mobility model. Therefore, we must redefine the `fixIfHostGetsOutside()` function by using the `handleIfOutside(REFLECT, targetPos, dummy, step, currentAngle)` function with the following parameters:

- `type`, which is an enum of the possible methods, we will choose `REFLECT`;
- `targetPos`, which is of type `Coord`, is the current target position;
- `dummy` is a dummy object type `Coord` (not used in the `REFLECT` mode);
- `step`, which is of type `Coord`, is the difference between the current and the target position. It should be computed and stored as a class variable for each movement step;
- `currentAngle`, which is a double, is the current moving angle. You should already have computed it for each movement step, just ensure that you store it as a class variable.

3.5 Simulation geometry

The simulation playground can be viewed as a (X, Y) orthonormated plan whose origin is located in the upper-left corner, higher abscissas being rightmost and higher ordinates on the bottom. As a consequence, this plan is indirect (in the sense of traditional trigonometry.)

Still, we will use the x-axis as reference of angles, and count positively the angles going from the x-axis to the y-axis. We will also use degrees as unit. For instance, the x-axis and the y-axis are separated by a right angle of measure 90° .

When aiming at a hotspot, a node can select its next angle within a given range, as explained in paragraph 2.3. We will approximate this range as being in the $[\alpha; \beta]$ range showed in figure 2. The most optimal angle (leading to the shortest route) being the angle θ , angle between the current position O and the center of the spot C .

Finding α and β means finding the angle between the current position O and the points A and B respectively. Those two points are defined as the intersection between the spot boundary and the line orthogonal to the line passing by the current position O and the spot center C .

To help you with your design, you are provided a template for the `HotSpotsMobility` module, which contains the following utility functions:

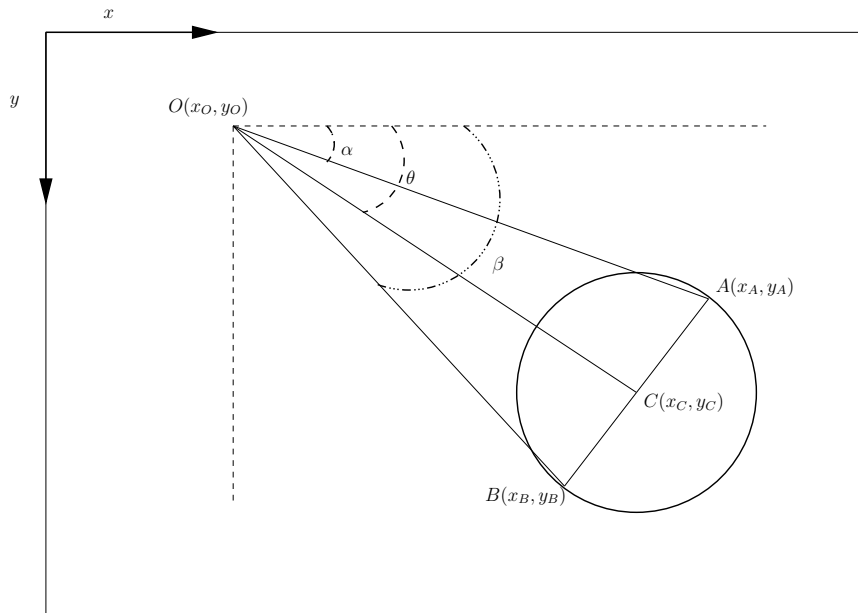


Figure 2: Hotspots playground geometry

- `getAngleFromCoords(Coord c1, Coord c2)`, which given the coordinates `c1` and `c2` of two points will return the angle between those points, as defined above, in the range $[0^\circ; 360^\circ]$.
- `getCoordFromAngle(Coord c, double d, double a)`, which given the coordinates of the starting point `c`, a distance `d` and an angle `a` will return the coordinates of the point which is at distance `d` of point `c` with angle `a`.

A simple composition of those functions will help you determine α and β and, thus, the range of angles allowed for the node movement.

The template is located in the `/home/user/Development/templates/lab04` folder.

4 Simulations with the hotspot mobility model

4.1 Testing

Once your implementation is complete and can be built, you will first test if your mobility model is following the requirements and works correctly.

For this purpose, use the `lab04-test.spots` and the `mobility-fw-test.ini` as configuration files. Copy them along with `positions.pl` from the `/home/user/Development/templates/lab04` directory to your current project directory.

Therefore compile your simulation for `cmdenv` output (using `make clean`, `opp_mf_makemake_cmd` then `make`).

Then launch a simulation run while redirecting the output to a trace file. Therefore, type:

```
./lab04 -r 1 > omnetpp.out
```

to redirect the output of the run 1 to the omnetpp.out file.

Then use the ./positions.pl script to parse the omnetpp.out file. This script will generate a gnuplot configuration file and use it to create a positions.eps graphic file plotting the spots location and size and the node positions during simulation. Your results should look like those presented in figure 1.

4.2 Simulation campaign

Once you have ensured that your mobility model performs correctly, you can launch a real simulation campaign of at least 20 runs using the lab04-campaign.spots and mobility-fw-campaign.ini files as configuration files.

As you built this model from the one used in previous lab, all the statistics collection features you have added should still be present and working.

Analyze and consolidate your results as you did in the previous lab to assess the impact of a close-to-reality mobility model on the protocol performance.

Next time:

Now that we have built a close-to-reality mobility pattern, we have improved the accuracy of our model to represent reality.

But in the real world, the presence of hotspots, which can be seen as classrooms or meeting rooms inside a building, or shops inside a mall, also has an impact on radio waves propagation. Real world hotspots are separated from the outside with walls and doors, and it is likely that nodes inside a given hotspot may more hardly receive data from outside nodes.

To take this effect into account, in next lab we will modify the propagation model of our simulation so as to get even closer to our use case reality.